ARTICLE TITLE: You've got the model-view-controller

Dr. John Hunt,
JayDee Technology Limited
Minerva House
Lower Bristol Road
Bath, BA2 9ER
Tel: 01225 789255
Fax: 0870 0548872
Email: john.hunt@jaydeetechnology.co.uk

Copy

Anyone who has ever tried to construct modular, object oriented user interfaces using the Java and Swing, knows how hard it can be. The result can easily end up being difficult to debug, complex to understand and maintain, and certainly not reusable (except by cutting and pasting!). However, huge benefits can be obtained by separating out the user interface (i.e. GUI components) from the application logic/code. This has been acknowledge for a long time and a number of approaches have been proposed over the years for separating the presentational aspect of an application from the logic of that application. In the case of client Java applications, used in multi-tier environments this is still true. In this column and the next, we will explore the use of the model-view-controller architecture/pattern (or just as the MVC for short). The MVC originated in Smalltalk but the concept has been used in many places. This article considers what the MVC is, why it is a good approach to GUI client construction and what features in Java support it.

1. What is the Model-View-Controller Architecture

With the advent of JDK 1.1 a new event model was introduced into Java. This event model is much cleaner than the previous approach and can result in simpler, clearer and more maintainable code. The introduction of this event model, along with existing Java facilities, allows the construction of modular user interfaces. In particular it allows the separation of the display of information, from the control or the user input to that display, as well as from the application. This separation is not a new idea and allows the construction of GUI applications that mirror the Model-View-Controller architecture. The intention of the MVC architecture is the separation of the user display, from the control of user input, from the underlying information model as illustrated

in Figure 1 [Krasner and Pope 1988]. There are a number of reasons why this is useful:

- Reusability of application and / or user interface components,
- Ability to develop the application and user interface separately,
- Ability to inherit from different parts of the class hierarchy.

Ability to define control style classes which provide common features separately from how these features may be displayed.

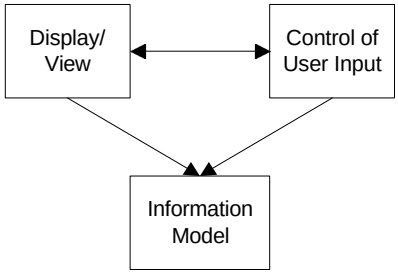

Figure 1: The Model-View-Controller architecture

This means that different interfaces can be used with the same application, without the application knowing about it. It also means that any part of the system can be changed without affecting the operation of the other. For example, the way that the graphical interface (the look) displays the information could be changed without modifying the actual application or how input is handled (the feel). Indeed the application need not know what type of interface is currently connected to it at all.

2. What Java facilities support the MVC

Java provides two facilities that together can allow the separation of the application, interface and control elements. These are:

- The observer / observable model. This allows application programs and user interfaces to be loosely coupled.
- The delegation event model. This provides listeners which act as controllers handling various events that may occur.

The use of the observer / observable mechanism is very powerful and has been used to create MVC style clients. However in this article we will focus on the use of the delegation event model. Why is? Essentially because more developers are familiar with the delegation event model and it can be used as successfully as the observer-Observable model to allow models to notify their views of the need to perform a redisplay style operation.

2.1 The Delegation Event Model

The Java delegation event model introduced the concept of listeners [Sevareid 1997]. Listeners are effectively objects that “listen” for a particular event to occur. When it does they react to it. For example, the event associated with the button might be that it has been “pressed”. The listener would then be notified that the button had been pressed and would decide

what action to take. This approach involves delegation because the responsibility for handling an event, generated by one object, may be another objects'.

The delegation model changed the way in which users created GUIs back in JDK 1.1. Using this model they defined the graphic objects to be displayed, added them to the display and associated them with a listener object. The listener object then handled the events that were generated for that object.

For example, if we wish to create a button which will be displayed on an interface and allow the user to exit without using the border frame buttons, then we would need to create a button and a listener for the action on the button:

```
exitButtonController = new ExitButtonController();
exitButton = new Button(" Exit ");
exitButton.addActionListener(exitButtonController);
```

This code creates a new user defined listener object, ExitButtonController, then creates a new button (with a label exit). It then adds the exitButtonController as the action listener for the button. That is, it is the object which will *listen* for action events (such as the button being pressed). The ExitButtonController class (presented below) provides a single instance method actionPerformed() which will initiate the System.exit(0) method.

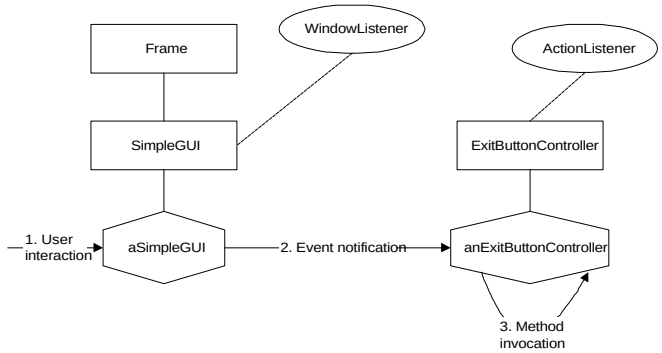

Figure 2: The class and object diagram using the delegation event model

The resulting class and instance structures are illustrated in Figure 2. As you can see from this diagram, the separation of interface and control is conceptually very clean.

The ExitButtonController class definition is presented below. There is of course no reason why you should call such classes controller, you could equally have called it an ExitButtonEventListener. However, the Listeners are the interface definitions. By choosing a different type of name, we make it clear we are talking about the classes intended to provide the execution control:

```
class ExitButtonController
        implements ActionListener {
  public void actionPerformed(ActionEvent event) {
```

```
            System.exit(0);
        }
    }
```

3. The MVC in Java

To provide a framework for the MVC within Java we can adopt a number of techniques, however the approach adopted here is illustrated by Figure 3. This diagram shows three interfaces (namely Controller, View and Model) that acts as markers for the core concepts or entities in the MVC framework. Note that the Controller and View interfaces define accessor methods for obtaining the model and either the view or controller respectively.

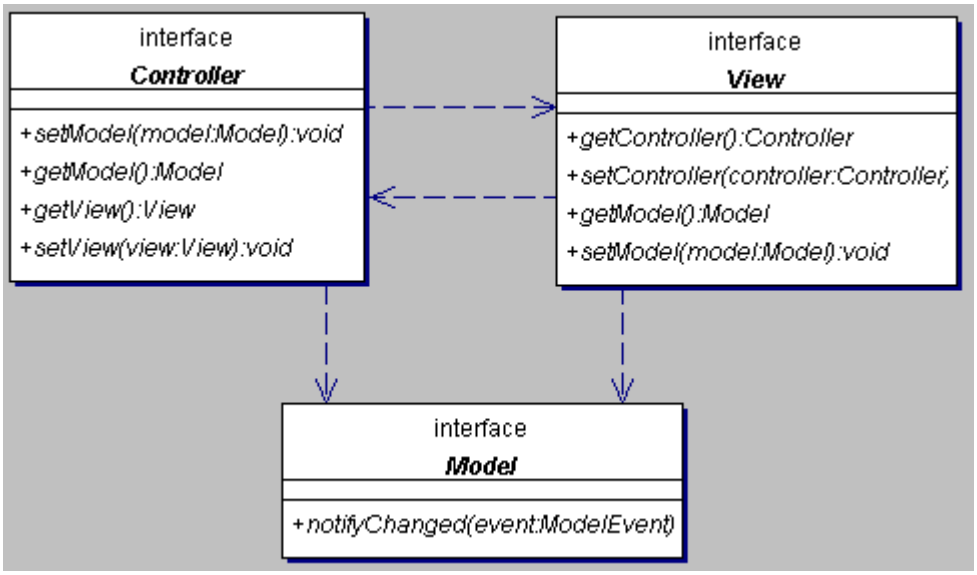

Figure 3: The interfaces defining the MVC entities

In Figure 3 the interface of most note is the Model interface. This interface contains just one method: the notifyChanged(ModelEvent event) method. This method will be used by all implementations of models to notify any interested objects of changes in the state of the Model. At this point all we know is that such a method will be provided and that the parameter to this method is ModelEvent.

This leads on nicely to Figure 4. This is the heart of the notification mechanism used within this implementation of the MVC. When a model wishes to notify interested objects of a change in its state. I must create a ModelEvent object (just as JButton creates an ActionEvent to notify a handler that a user has clicked on it). In this cases the ModelEvent object has been made a subclass of ActionEvent however it does not need to subclass this class – it could subclass any Event object as identified by a designer. The ModelEvent class adds an additional property to those inherited, this property allows a ModelEvent to hold an amount value (this is really tying too closely to the calculator application but keeps things simpler later on).

As we are defining our own event class, we also need to define a listener interface for objects that wish to be notified of ModelEvent. The ModelListener

interface does this. Note that the JFrameView class (the root of all windows that wish to act as the view element of a MVC framework) implements the ModelListener interface. thus all JFrameViews (and its subclasses) can be notified of ModelEvents.

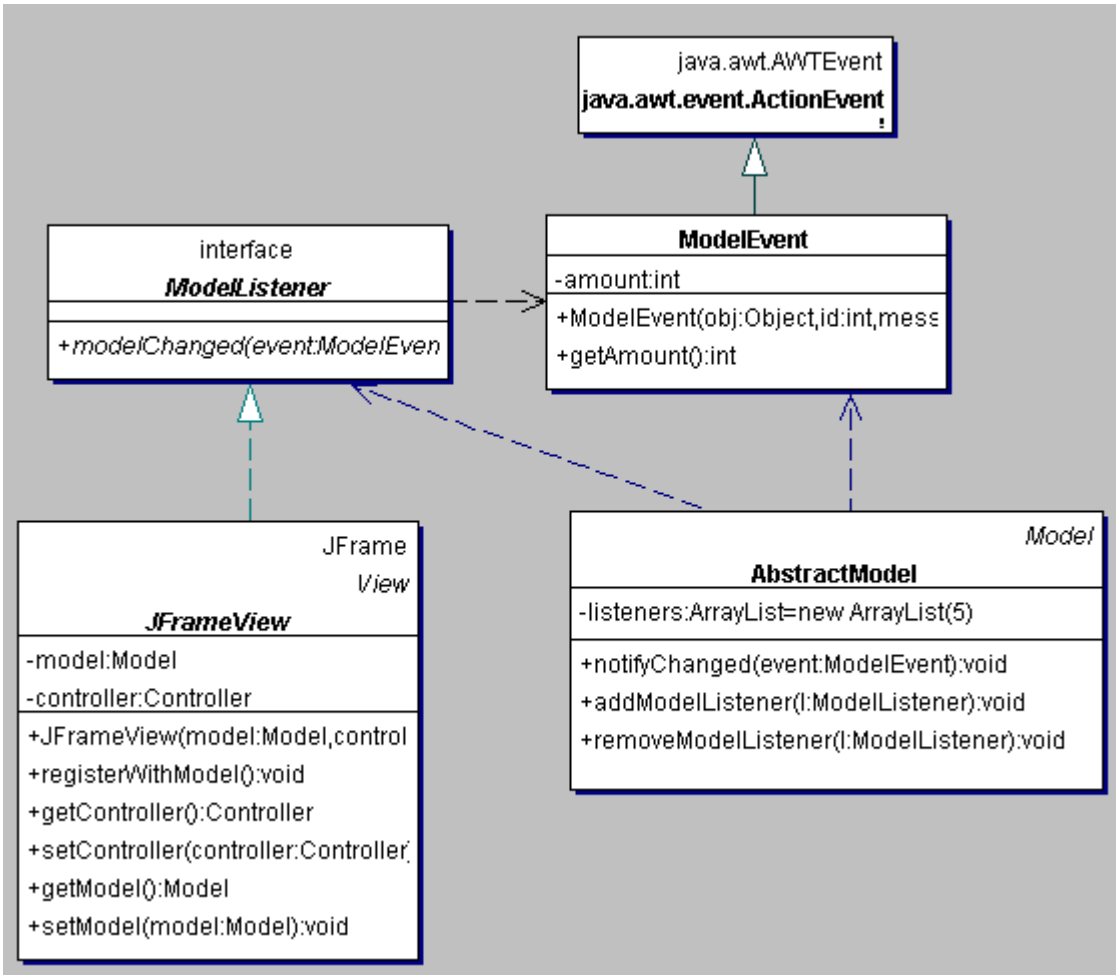

Figure 4: The event notification aspects of the framework

Finally the AbstractModel class provides an implementation for notifyChanged that takes a ModelEvent and sends that event in turn to each of the listeners registered with the model. Views (and any other listener objects) can register with any subclasses of AbstractModel via the addModelListener method that takes an object whose class implements the ModelListener interface as a parameter. Objects can remove themselves from listener to the models events via the removeModelListener method.

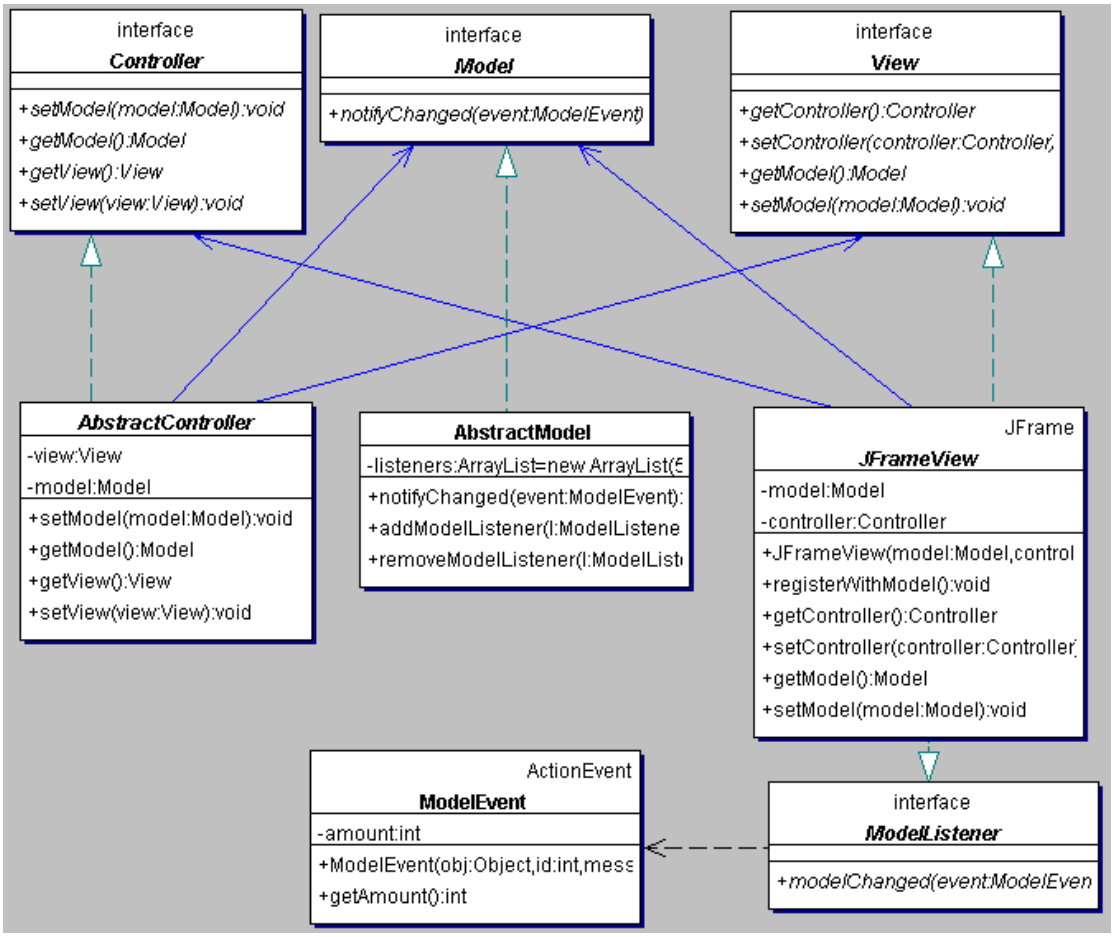

Figure 5: The whole generic MVC framework

Figure 5 illustrates the final generic MVC framework as implemented in this column. Note that each of the root classes (AbstractController, AbstractModel and JFrameView) reference the associated interfaces rather than the classes. This helps maintain the generic nature of the framework.

4. A simple calculator application

To illustrate how this framework may be used in a simple application we shall construct a very simple calculator application as illustrated in Figure 6. This application only allows integer addition or subtract for value sin the range 0 to 9. Thus it is not possible to enter the value 10 – this is to keep the application *very* simple.

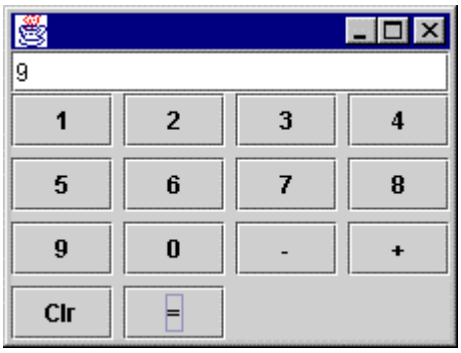

Figure 6: The GUI for the calculator application

The overall structure of the application is that illustrated in Figure 7. Note that the view object and the controller objects have inherited links that allow them to communicate. However, although the interface and the controller objects have links to the application model (CalculatorModel), the application model knows nothing directly about the view or the controllers. This means that the application logic in the calculator is independent of the view and its controllers and may actually have various different GUI interfaces associated with it. One of the advantages of this approach is that any one of the three elements can be modified without the need to change the others.

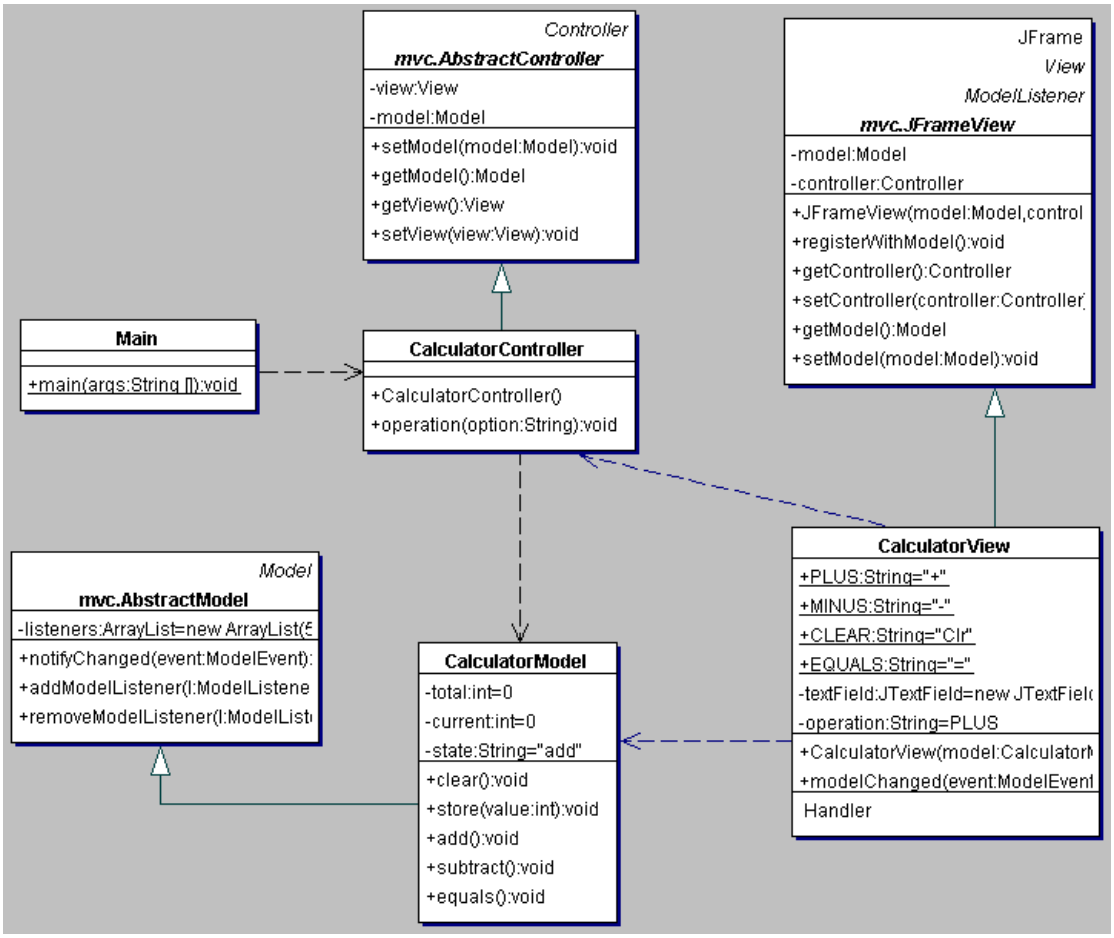

Figure 7: The MVC architecture as it is implemented by the account application

The system interaction is illustrated in Figure 8. This figure illustrates the various messages sent once a user clicks on the deposit button. There are a number of points you should note about this example:

1. Neither the display nor the controller hold onto the balance. It is obtained from the account whenever it is needed.
2. The controller relies on the delegation event model to determine that it should do something.
3. When the controller asks the model to change it doesn't tell the display – the display finds out about the change through the observer / observable mechanism.

4. The account is unaware that the message deposit(amount); came from the controller. Thus any object could send a deposit message and the account would still inform its dependents about the change.

4.1 Swing component event handling

In the approach being described in this column the event handling code resides within the View classes (as it is essentially a swing related operation and all swing related functionality should sit within the View). This means that an event handler (for example an inner classes) implements the actionPerformed method but then calls a method on the controller that will determine what should happen next.

The controller in turn receives the information provided by the actionPerformed method of the event handler and now determines what action should happen next. In this particular application it determines whether the model should be notified of an operation such as addition, subtraction, clear or equals or that the model should be passed the integer entered by the user. However it does not more than this.

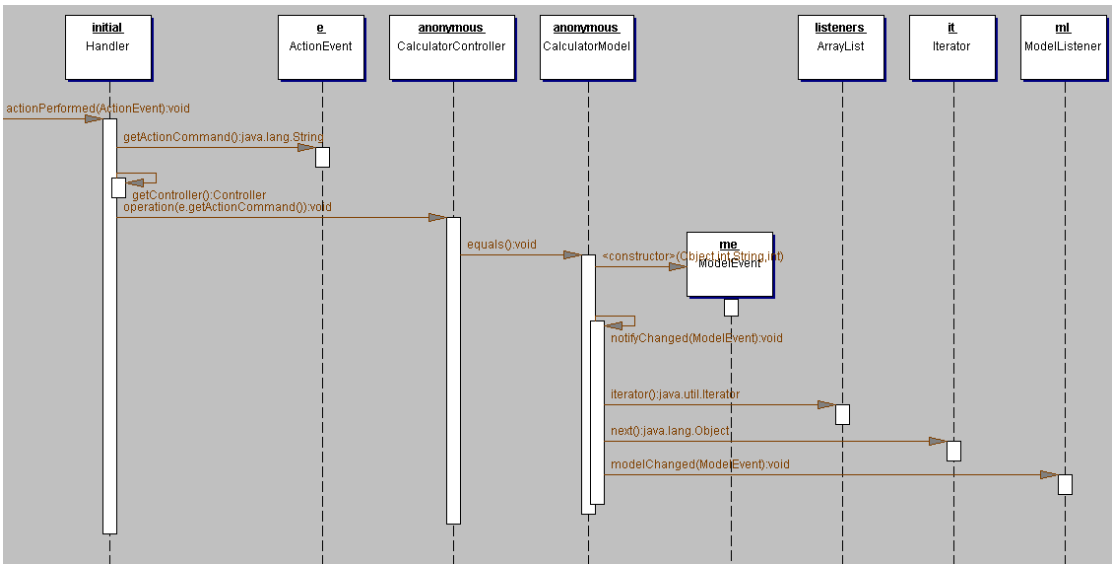

Figure 8: Interaction in the MVC

In turn the application model (the CalculatorModel) handles what the behaviour of each of the add, subtract, clear and equals methods should be. Once it has done this, it creates a ModelEvent and calls notifyChanged passing the ModelEvent in. This method, inherited form the parent class, then notifies the view that it must update itself via the modelChanged method. In our case the view obtains the new value to display form the event and displays it within the JTextField at the top of the CalculatorView.

4.2 Frames, Panels and Layout Managers

The interface object is, of course, made up of a number of objects such as a frame, a number of panels, as well as graphic components such as buttons and text fields. In turn layout managers are used to control the way in which these objects are arranged within the window frame. An interesting point to note is that the exit button controller is used without any modification from

previous examples. In addition the abstract buttonController class (from which the buttonPanelController class inherits) is a reusable class for any object acting as a controller within an MVC style architecture.

4.3 The Application Code

The source code for this application builds upon the generic MVC framework defined earlier – which involves more work the first time this framework is followed, but the overhead is reduced for future applications.

5. Discussion

In this column I have tried to describe a way of constructing graphical user interface applications (and of course applets) that is robust, principled and reusable. It allows the various classes to inherit from different parts of the class hierarchy, to implement different interfaces and to provide clearly defined functionality. All of which lead to clearer, more comprehensible code. It can be seen that such an approach allows the GUI to be structured in an object-oriented manner. It would, of course, be possible to place all of the application within a single class. This class would hold the application code, the window definition and the event handling code. However, we would have lost the following advantages:

- Reusability of parts of the system,
- The ability to inherit from different parts of the class hierarchy,
- Modularity of system code,
- Resilience to change,
- Encapsulation of the application.

Although these issues might not be a problem for an application as simple as that presented here, for real world systems they would certainly be significant. It is hoped that you are now aware of the benefits of adopting the MVC architecture and will try to adopt this approach in your own systems.

In the next column we will consider a modified version of the MVC framework, referred to as the Hierarchical MVC that allows more complex, and realistic, applications to be constructed.

6. References

[Krasner and Pope 1988] G. E. Krasner and S. T. Pope, A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80, *JOOP* 1(3), pp. 26-49, 1988.

[Sevareid 1997] Jeremy Sevareid, The JDK 1.1s New Delegation Event Model, *Java Report*, pp. 59 – 79.

Listings

Listing 1: The Model interface

```
package mvc;

/**
 * This interface must be implemented by all class that wish to play the Model
```

```
 * role within the MVC framework.
 *  <p>
 *  The only method specified by the interface is the <code>notifyChanged()
 * </code> method. */
public interface Model {
    void notifyChanged(ModelEvent event);
}
```

Listing 2: The Model interface

```
package mvc;
/**
 * The Controller interface is the interface which must be implemented by
 * all classes which wish to take the role of a Controller.
 * All controllers must be able to reference a model and a view object.
 * <p>
 * The primary role of a Controller within the MVC is to determine what
 * should happen in response to user input.
 */
public interface Controller {
    void setModel(Model model);
    Model getModel();
    View getView();
    void setView(View view);
}
```

Listing 3: The View interface

```
package mvc;
/**
 * This interface must be implemented by all classes that wish to take the role
 * of the View within the MVC framework.
 * The role of a View is the display of information and the capture of
 * data entered.
 */
public interface View {
    Controller getController();
    void setController(Controller controller);
    Model getModel();
    void setModel(Model model);
}
```

Listing 4: The AbstractModel class

```
package mvc;
import java.util.ArrayList;
import java.util.Iterator;

/**
 * Abstract root class of Model hierarchy - provides basic
 * notification behaviour
 */
public abstract class AbstractModel implements Model {
   private ArrayList listeners = new ArrayList(5);
   /**
    * Method that is called by subclasses of AbstractModel when they want to
    * notify other classes of changes to themselves.
    */
   public void notifyChanged( ModelEvent event ){
      ArrayList list = (ArrayList)listeners.clone();
            Iterator it = list.iterator();
            while (it.hasNext()) {
```

```
                ModelListener ml = (ModelListener)it.next();
                    ml.modelChanged(event);
                }
        }

        /**
         * Add a ModelListener to the list of objects interested in ModelEvents.
         */
        public void addModelListener(ModelListener l) {
            listeners.add(l);
        }

        /**
         * Remove a ModelListener from the list of objects interested in ModelEvents
         */
        public void removeModelListener(ModelListener l) {
            listeners.remove(l);
        }
}
```

Listing 5: The AbstractController class

```
package mvc;
/**
 * The root of the Controller class hierarchy is the AbstractController class.
 * This class defines all the basic facilities required to implement a
 * controller. That is, it allows a view and model to be linked to the
 * controller.
 * <p>
 * It also provides a set of constructors and set and get methods for views and
 * models
 */
public abstract class AbstractController implements Controller {
    private View view;
    private Model model;

    public void setModel(Model model) {
        this.model = model;
    }

    public Model getModel() {
        return model;
    }

    public View getView() {
        return view;
    }

    public void setView(View view) {
        this.view = view;
    }
}
```

Listing 6: The JFrameView class

```
package mvc;
import javax.swing.*;

/**
 * The JFrameView class is the root class of the view class hierarchy for top level
 * (swing) frames. It allows a controller and a model to be registered and can register
```

```
 * itself with a model as an observer of that model.
 * <p>
 * It this extends the JFrame class.
 * <p>
 * It requires the implementation of the <code>modelChanged(ModelEvent event);</code>
 * method in order that it can work with the notification mechanism in Java.
 */
abstract public class JFrameView extends JFrame implements View, ModelListener{
    private Model model;
    private Controller controller;
    public JFrameView (Model model, Controller controller) {
        setModel(model);
        setController(controller);
    }
    public void registerWithModel() {
            ((AbstractModel)model).addModelListener(this);
    }
    public Controller getController() {
        return controller;
    }
    public void setController(Controller controller) {
        this.controller = controller;
    }
    public Model getModel() {
        return model;
    }
    public void setModel(Model model) {
        this.model = model;
        registerWithModel();
    }
}
```

Listing 7: The ModelEvent class

```
package mvc;
import java.awt.event.ActionEvent;

/**
 * Used to notify interested objects of changes in the
 * state of a model
 */
public class ModelEvent extends ActionEvent {
   private int amount;
   public ModelEvent( Object obj, int id, String message, int amount){
      super( obj, id, message ) ;
      this.amount = amount;
   }
   public int getAmount() {
           return amount;
   }
}
```

Listing 8: The ModelListener class

```
package mvc;
public interface ModelListener {
    public void modelChanged(ModelEvent event);
}
```

Listing 9: The Main class

```
package calculator;
```

```
public class Main {
   public static void main(String [] args) {
     new CalculatorController();
   }
}
```

Listing 10: The CalculatorController class

```
package calculator;
import mvc.*;
public class CalculatorController extends AbstractController {
    public CalculatorController() {
       setModel(new CalculatorModel());
       setView(new CalculatorView((CalculatorModel)getModel(),
                                          this));
       ((JFrameView)getView()).setVisible(true);
    }
    public void operation(String option) {
       if (option.equals(CalculatorView.MINUS)) {
            ((CalculatorModel)getModel()).subtract();
       } else if (option.equals(CalculatorView.PLUS)) {
            ((CalculatorModel)getModel()).add();
       } else if (option.equals(CalculatorView.CLEAR)) {
            ((CalculatorModel)getModel()).clear();
       } else if (option.equals(CalculatorView.EQUALS)) {
            ((CalculatorModel)getModel()).equals();
       } else {
            ((CalculatorModel)getModel()).store(Integer.parseInt(option));
       }
    }
}
```

Listing 11: The CalculatorModel class

```
package calculator;
import mvc.*;

public class CalculatorModel extends AbstractModel {
    private int total = 0;
    private int current = 0;
    private String state = "add";
    public void clear() {
            total = 0;
            store(0);
    }
    public void store(int value) {
        current = value;
        ModelEvent me = new ModelEvent(this, 1, "", current);
        notifyChanged(me);
    }
    public void add() {
            state = "add";
            total = current;
    }
    public void subtract() {
            state = "subtract";
            total = current;
    }
    public void equals() {
            if (state=="add") {
               total += current;
```

```
            } else {
                total -= current;
            }
            current = total;
            // now notoify any interested parties in the new total
            ModelEvent me = new ModelEvent(this, 1, "", total);
            notifyChanged(me);
    }
}
```

Listing 12: The CalculatorView class

```
package calculator;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import mvc.*;

public class CalculatorView extends JFrameView {
    public static final String PLUS = "+";
    public static final String MINUS = "-";
    public static final String CLEAR = "Clr";
    public static final String EQUALS = "=";
    private JTextField textField = new JTextField();
    private String operation = PLUS;

    public CalculatorView(CalculatorModel model, CalculatorController controller) {
        super(model, controller);
        textField.setText("0");
        this.getContentPane().add(textField, BorderLayout.NORTH);
        JPanel buttonPanel = new JPanel();
        Handler handler = new Handler();
        JButton jButton1 = new JButton("1");
        jButton1.addActionListener(handler);
        JButton jButton2 = new JButton("2");
        jButton2.addActionListener(handler);
        JButton jButton3 = new JButton("3");
        jButton3.addActionListener(handler);
        JButton jButton4 = new JButton("4");
        jButton4.addActionListener(handler);
        JButton jButton5 = new JButton("5");
        jButton5.addActionListener(handler);
        JButton jButton6 = new JButton("6");
        jButton6.addActionListener(handler);
        JButton jButton7 = new JButton("7");
        jButton7.addActionListener(handler);
        JButton jButton8 = new JButton("8");
        jButton8.addActionListener(handler);
        JButton jButton9 = new JButton("9");
        jButton9.addActionListener(handler);
        JButton jButton0 = new JButton("0");
        jButton0.addActionListener(handler);
        JButton minusButton = new JButton(MINUS);
        minusButton.addActionListener(handler);
        JButton plusButton = new JButton(PLUS);
        plusButton.addActionListener(handler);
        JButton clearButton = new JButton(CLEAR);
        clearButton.addActionListener(handler);
        JButton equalsButton = new JButton(EQUALS);
        equalsButton.addActionListener(handler);
```

```
        buttonPanel.setLayout(new GridLayout(4, 4, 5, 5));
        this.getContentPane().add(buttonPanel, BorderLayout.CENTER);
        buttonPanel.add(jButton1, null);
        buttonPanel.add(jButton2, null);
        buttonPanel.add(jButton3, null);
        buttonPanel.add(jButton4, null);
        buttonPanel.add(jButton5, null);
        buttonPanel.add(jButton6, null);
        buttonPanel.add(jButton7, null);
        buttonPanel.add(jButton8, null);
        buttonPanel.add(jButton9, null);
        buttonPanel.add(jButton0, null);
        buttonPanel.add(minusButton, null);
        buttonPanel.add(plusButton, null);
        buttonPanel.add(clearButton, null);
        buttonPanel.add(equalsButton, null);
        pack();
    }

    // Now implement the necessary event handling code
    public void modelChanged(ModelEvent event) {
            String msg = event.getAmount() + "";
            textField.setText(msg);
    }

    // Inner classes for Event Handling
    class Handler implements ActionListener {
        // Event handling is handled locally
        public void actionPerformed(ActionEvent e) {
            ((CalculatorController)getController()).operation(e.getActionCommand());
        }
    }
}
```